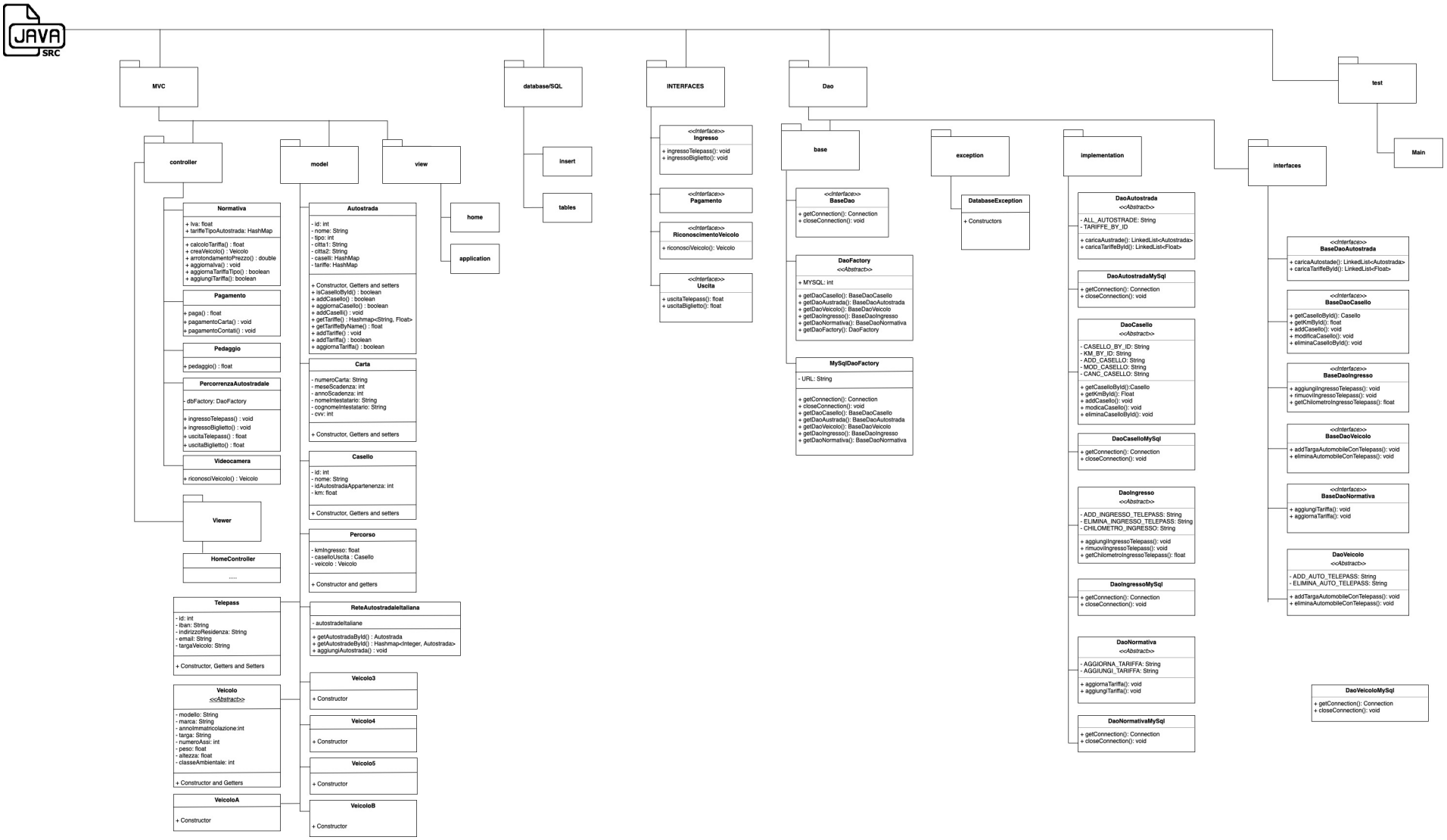
OOSD Project

requisiti | 1.1

- Gestione Ingresso e Uscita di un automobile (con telepass o con biglietto)
- Calcolo del pedaggio
- Predisposizione del sistema alla gestione del pagamento
- Predisposizione all'eventuale aggiunta di metodi di calcolo del pedaggio
- Interfacciamento del sistema con il Database

modello dell'architettura software | 2.1



descrizione dell'architettura | 2.2
descrizione delle scelte | 2.3

La progettazione è stata regolata dall'utilizzo del pattern MVC che andremo ad illustrare in breve.

Questo pattern è suddiviso in tre parti, il Model, il View ed il Controller.
Il **model** conterrà le classi che rappresentano la parte dei dati con i relativi metodi di lettura e scrittura su essi (get e set). Interpreta il comportamento dell'applicazione in termini di dominio del problema, in maniera del tutto indipendente dall'interfaccia utente. Il modello gestisce i dati, la logica e le regole dell'applicazione.

Come **view** si intende un qualsiasi tipo di rappresentazione in output. Tipiche nelle strutture MVC sono le viste multiple delle stesse informazioni.

Infine la parte **controller** conterrà una serie di applicazioni che andranno a modificare il modello. Praticamente stabilisce i comandi per il model e la view, facendoli interagire tra loro.

Nel Model abbiamo scelto di inserire le seguenti Classi :

- Autostrada

  questa classe contiene gli attributi caselli, cittàUno, cittàDue, id, nome, tariffe e tipo.
  Gli attributi caselli e tariffe rappresentano rispettivamente la lista di caselli contenuti dall'autostrada e le tariffe legate all'autostrada divise per classe veicolare. Questi due attributi sono stati implementati nel seguente modo.

```
private HashMap<Integer, Casello> caselli = new HashMap<Integer, Casello>();
   private HashMap<String, Float> tariffe = new HashMap<String, Float>();
```

  È stata fatta questa scelta in modo da rendere l'accesso ai dati molto più veloce tramite le chiavi delle hashmap, soprattutto nel caso dei caselli essendo numerosi. Nell'attributo caselli la chiave è rappresentata dall'id di ogni uno di essi, mentre nell'attributo tariffe è rappresentata dal nome della classe veicolare sotto forma di stringa.
  In questa classe, oltre ai metodi get e set standard sono stati previsti i seguenti metodi aggiuntivi.

  Metodo che restituisce la lista di caselli sotto forma di Set in modo da prestarsi ad utilizzi differenti

```
public Set<Map.Entry<Integer, Casello>> getCaselliSet() { return this.caselli.entrySet(); }
```

  Metodo che restituisce un casello dato iil suo id

```
public Casello getCaselloById(int id) { return this.caselli.get(id); }
```

Metodo che controlla se un casello è presente nella lista restituendo un risultato buleano.

```
public boolean isCaselloById( int id ) { return this.caselli.containsKey(id); }
```

Metodo che aggiunge un casello controllando se esso è già presente nell'autostrada.

```
public boolean addCasello(Casello nuovoCasello) {
      if ( ( isCaselloById(nuovoCasello.getId()) ) ) return false;  // casello gi presente
      this.caselli.put(nuovoCasello.getId(), nuovoCasello);
      return true;
}
```

Metodo che aggiorna un casello passando il nuovo casello da sostituire al vecchio. l'aggiornamento avverrà con successo se il casello inserito ha lo stesso id di uno dei caselli già presenti.

```
public boolean aggiornaCasello(Casello nuovoCasello) {
      if ( !( isCaselloById(nuovoCasello.getId()) ) ) return false;  // casello non presente
      this.caselli.put(nuovoCasello.getId(), nuovoCasello);
      return true;
}
```

Metodo che permette l'inserimento dei caselli tramite un oggetto di tipo List.

```
public void addCaselli( List<Casello> caselli ) {
     for (Casello c : caselli){
         this.caselli.put(c.getId(), c);
     }
}
```

Metodo che restituisce il valore di una tariffa data la stringa rappresentante la classe veicolare.

```
public float getTariffaByName(String nomeTariffa) { return this.tariffe.get(nomeTariffa); }
```

Metodi di aggiunta e aggiornamento di una tariffa che funzionano con lo stesso criterio di quelli mostrati per i caselli

```
public boolean addTariffa(String classeVeicolo, float nuovaTariffa) {
     if ( this.tariffe.containsKey(classeVeicolo) ) return false;  // tariffa gia presente
     this.tariffe.put(classeVeicolo, nuovaTariffa);
     return true;
}
public boolean aggiornaTariffa(String classeVeicolo, float nuovaTariffa) {
     if ( !( this.tariffe.containsKey(classeVeicolo) ) ) return false;  // tariffa non presente
     this.tariffe.put(classeVeicolo, nuovaTariffa);
     return true;
```

```
}
```

- Carta

  questa classe contiene gli attributi annoScadenza, cognomeIntestatario, nomeIntestatario, cvv, meseScadenza e annoScadenza (attributi necessari per effettuale il pagamento).

  I metodi che troviamo all'interno di questa classe sono i classici get. I set non sono stati previsti in quanto una volta istanziato l'oggetto Carta questo verrà utilizzato per il pagamento e poi cancellato perciò non ci sarà ne modo ne motivo di modificarlo in un secondo momento alla creazione.

- Casello

  questa classe contiene gli attributi id, chilometro, nome e autostradaAppartenenza.

  I metodi che troviamo sono i classici get e set.

- Percorso

  questa classe rappresenta il percorso di un veicolo dal casello di ingresso (chilometro) al casello di uscita, perciò gli attributi che contiene sono chilometroIngresso, caselloUscita e veicolo.

  I metodi che troviamo in questa classe sono i classici get. I set non sono presenti in quanto l'oggetto di tipo Percorso verrà istanziato al momento dell'arrivo dell'automobile al casello di uscita e verrà passato al metodo che calcolerà il pedaggio. La comodità di questa classe è quella di raccogliere tutte le informazioni relative al calcolo del pedaggio in un unico oggetto in modo tale da rendere il codice modulare. Se, ad esempio, verrà introdotto

l'attributo di indice di inquinamento acustico per il calcolo del pedaggio, questo cambiamento coinvolgerà solamente la classe Automobile (con l'aggiunta dell'attributo) e la classe Pedaggio (con l'aggiunta del metodo

che calcola il nuovo pedaggio) mentre i metodi potranno continuare a passare come argomento solamente l'oggetto Percorso in quanto conterrà l'oggetto Veicolo e al suo interno troveremo gli attributi necessari al calcolo del pedaggio.

- ReteAutostradaleItaliana

  Questa classe contiene una lista di tutte le autostrade italiane implementata attraverso un hashmap dove le chiavi sono gli id di ogni autostrada.

- Telepass

  Questa classe contiene gli attributi email, iban, id, indirizzoResidenza, targaVeicolo e i metodi get. Per lo stesso motivo della classe Carta non sono previsti i metodi set.

- Veicolo

  Questa è una classe astratta che contiene gli attributi altezza, annoImmatricolazione, classeAmbientale, marca, modello, numeroAssi, peso e targa.

  È stato scelto di fare questa classe astratta in modo da creare una specializzazione per ogni tipo di veicolo. Le sottoclassi (Veicolo1A, Veicolo1B, Veicolo13, Veicolo14, Veicolo15) non faranno altro che ereditare i metodi e gli attributi di veicolo. A prima vista può sembrare una scelta poco sensata ma in caso di un'aggiramento futuro ci

potranno essere normative che prevederanno un aggiunta di attributi differenti a seconda del tipo di veicolo e questa scelta si rivelerà di fondamentale importanza.

Nel Controller abbiamo scelto di inserire le seguenti classi.

- Normativa

  Questa classe è stata pensata per raccogliere tutti i dati e le istruzioni che sono regolate da delle normative in vigore che potrebbero cambiare nel tempo e su cui si basa sia la differenziazione dei veicoli per classi sia l'iva sia le tariffe relative al tipo di autostrada (pianura o montagna).

  Metodo che calcola la tariffa relativa ad un veicolo tenendo conto della classe di quest'ultimo e dell'autostrada che si sta percorrendo.

  ```
  public static float calcoloTariffa(Veicolo veicolo, Autostrada autostrada)
  ```

  Metodo che si occupa dell'arrotondamento del risultato finale ed è stato messo all'interno di normativa in quanto potrebbe cambiare da un momento all'altro la legge sull'arrotondamento finale del pedaggio.

  ```
  public static double arrotondamentoPrezzo(double prezzo)
  ```

  Metodo che aggiorna il valore dell'IVA

  ```
  public static void aggiornaIva(float newIva)
  ```

Metodi che occupano dell'aggiunta e dell'aggiornamento delle tariffe relative al tipo di autostrada.

```
public static boolean aggiornaTariffe(int tipo, float newTariffa) {
     if ( !(tariffeTipoAutostrada.containsKey(tipo)) ) return false;
     tariffeTipoAutostrada.put(tipo, newTariffa);
     return true;
}
public static boolean aggiungiTariffa(int tipo, float newTariffa) {
     if ( tariffeTipoAutostrada.containsKey(tipo) ) return false;
     tariffeTipoAutostrada.put(tipo, newTariffa);
     return true;
}
```

- Pagamento

  Questa classe si occupa del pagamento e i suoi metodi verranno chiamati subito dopo il calcolo del pedaggio.
  Il metodo principale all'interno di questa classe è il metodo

  ```
  public static float paga(float prezzo)
  ```

che in un caso reale riconoscerà se l'utente sta introducendo dei contanti o una carta e chiamerà il metodo addetto a gestire il pagamento nel modo adatto.

- Pedaggio

  In questa classe troviamo il metodo che si occupa del calcolo del pedaggio. È stato scelto di creare un classe apposita per questa operazione in quanto ci potrebbero essere molte riforme che prevederanno un calcolo del pedaggio fatto in diversi modi. Con l'utilizzo di una classe apposita, come nel nostro caso, i metodi verranno aggiunti tutti al suo interno e basterà cambiare il nome del metodo chiamato nelle classi controller che si occupano di reperire i dati da passare poi a questo metodo. All'interno di questa classe è presente anche un metodo commentato che funge da esempio per ciò che è stato appena spiegato.

- Percorrenza Autostradale

  Questa classe implementa le interfacce Ingresso ed Uscita che descriveremo in seguito nello specifico. L'obiettivo della classe PercorrenzaAutostradale è quella di occuparsi di tutte le operazioni riguardanti la percorrenza dell'autostrada da casello di ingresso a casello di uscita di ogni veicolo. I comportamenti saranno forniti alla classe tramite l'implementazione di interfacce apposite. Nel nostro caso i comportamenti ereditati sono sia quello di ingresso di un automobile sia quello di uscita, entrambi distinti per telepass e biglietto.
  I metodi che troviamo sono i seguenti.

```
public void ingressoTelepass(Casello caselloIngresso, String targa) throws DatabaseException
```

```
public float uscitaTelepass(Casello caselloUscita, String targa) throws DatabaseException,
                            FileNotFoundException, IOException
```

```
public void ingressoBiglietto(Casello caselloIngresso) throws IOException
```

```
public float uscitaBiglietto(Casello caselloUscita, String targa) throws IOException
```

Analizziamoli brevemente uno ad uno.

ingressoTelepass : questo metodo si occupa di salvare nel database l'associazione tra la targa dell'automobile che entra e il casello nel quale è' entrata. Abbiamo fatto questo tipo di scelta in quanto, in caso di guasto del sistema, gli automobilisti dotati di biglietto, arrivando al casello di uscita potranno fornire, tramite quest'ultimo, le informazioni necessarie a risalire al percorso effettuato. Gli automobilisti che sono entrati dalla corsia telepass, invece, non hanno nulla per risalire al loro percorso fatto perciò facciamo questo tipo di "salvataggio" dei dati in memoria.

ingressoBiglietto : questo metodo creerà un file .txt che simulerà il biglietto cartaceo rilasciato dal casello. Come illustrato dalla specifica, questo file verrà preso in input dal casello di uscita.

uscitaTelepass : questo metodo si occupa di reperire dal database il chilometro corrispondente al casello di ingresso. Questi dati sono collegati alla targa dell'automobile che il metodo prenderà in input. Successivamente cancellerà l'associazione inserita con il metodo ingressoTelepass in quanto, una volta che il veicolo è uscito dall'autostrada non avrà più senso mantenere in memoria questi dati. Verrà poi creato l'oggetto Percorso che verrà passato al metodo pedaggio e verrà infine restituito il prezzo risultante dalla chiamata a quest'ultimo metodo.

uscitaBiglietto : questo metodo legge il file precedentemente creato da ingressoBiglietto da cui prende l'id del casello di ingresso. Successivamente verrà creato l'oggetto Percorso che verrà passato al

metodo pedaggio e verrà infine restituito il prezzo risultante dalla chiamata a quest'ultimo metodo.

- Videocamera

  Questa classe contiene un unico metodo che si occupa del riconoscimento del veicolo tramite una videocamera fisica.

  ```
  public Veicolo riconosciVeicolo(String targa) throws FileNotFoundException, IOException
  ```

  Essendo impossibilitati ad utilizzare una videocamera abbiamo preparato dei file di testo collocati in test.libretti che vanno a simulare il libretto a cui il sistema reale accede dopo aver letto la targa. La scelta di quale automobile utilizzare verrà fatta dall'interfaccia grafica tramite la targa. Il metodo, inoltre restituirà l'oggetto Automobile creato con i dati appena reperiti. La creazione dell'oggetto sarà incaricata al metodo creaVeicolo della classe Normativa analizzato in precedenza.

- HomeController

  Questa classe è il controller della parte grafica che si occupa di mettere in comunicazione la parte view con i model e i controller. Nella classe troviamo le definizioni degli oggetti che compongono la parte grafica e i metodi che vengono avviati tramite un azione fatta sull'interfaccia dall'utente.

Nel View troviamo solamente due file che andremo a descrivere in breve.

- home.fxml è il primo file che contiene il codice generato da sceneBuilder. Questo programma è stato utilizzato per semplificare e velocizzare la creazione della parte grafica.

- il secondo file è application.css ed è un file di tipo Cascade Style Sheets. non è stato messo niente riguardante la grafica all'interno di esso ma è già importato nell'esecuzione del programma in modo da renderlo pronto all'utilizzo.

L'interfaccia che il programma ci metterà davanti è questa ed è caratterizzata dai componenti che ora andremo a descrivere.

In alto a sinistra troviamo una serie di 6 Button che serviranno a selezionare l'automobile per la simulazione tramite la sua targa. Nella zona sottostante ai Button appena descritti troviamo una Label di visualizzazione delle informazioni sull'automobile dell'automobile selezionata. Sulla parte destra, partendo dall'alto, troviamo il Button di chiusura dell'applicazione, quattro menu a tendina e il tasto di start. I quattro menu sono

rispettivamente adibiti alla scelta dell'autostrada, scelta del casello di ingresso, scelta del casello di uscita e scelta del tipo di ingresso (telepass o biglietto). Come si può vedere dalla foto allegata, il tasto start ed i menu a tendina sono bloccati e si sbloccano sequenzialmente a partire dalla scelta dell'automobile. Al click del tasto di start si aprirà una finestra popup con l'importo da pagare. Inoltre abbiamo Previsto i 2 Button che permettono di modificare, aggiungere o eliminare caselli.

Oltre ai package MVC ne sono previsti altri che andremo ad illustrare.

- Interfaces

  che conterrà le interfacce

- database.SQL

  che conterrà i file .SQL per la creazione e il riempimento della base di dati

- test

  che conterrà la classe principale di test dell'applicazione nella quale vengono create le classi a partire dai dati presenti nel database e che conterrà il metodo main.

- test.biglietti

  che conterrà i biglietti delle auto che simulano il biglietto cartaceo rilasciato dai caselli.

- test.libretti

  che conterrà i libretti di simulazioni delle automobili.

D.A.O

Il D.A.O. (data access object) è un design pattern organizzato in classi e interfacce che permette l'interfacciamento con il database tramite le query. Grazie al dao, dai controller eviteremo di lanciare delle query direttamente ma ci limiteremo a chiamare dei metodi che le lanceranno e ci restituiranno gli eventuali risultati. Il vantaggio di usare questo design pattern sono i seguenti :

- La classe BaseDao ci permette di scegliere il tipo di DBMS da utilizzare, nel nostro caso è stato previsto soltanto MySql ma è stata fatta un implementazione tale da poter aggiungere diversi DBMS con il minimo impatto sul codice.
- Il codice nei controller risulta molto più pulito e leggibile in quanto andiamo ad istanziare oggetti dao specifici per l'esecuzione di determinate query, perciò leggendo solamente il nome dell'oggetto dao ed il metodo chiamato si può facilmente capire quale sarà l'effetto della query che verrà eseguita.
- Nel caso in cui si cambierà il tipo di DataSource utilizzato, i metodi del controller non dovranno essere cambiati in quanto si dovrà intervenire soltanto sui metodi del dao che sono strutturati in modo da essere facilmente reperibili e limitare l'impatto di modifica di codice a cascata in quanto indipendenti dal resto.
- Facilità nella lettura e nella ricerca di parti di codice che ci occorrono in quanto ogni package, classe e interfaccia che lo compone ha un nome e una posizione molto intuitiva. Cercare le medesime cose all'interno dei controller sarebbe molto più confusionario.

L'architettura software del D.A.O. è riportata nello schema UML sovrastante, perciò da li si può capire per bene come sono state organizzare le interfacce e le classi. Nonostante lo schema preferiamo fare una breve descrizione dei package che compongono questo design pattern.

- base

  contiene le parti base del D.A.O

  - BaseDao : l'interfaccia che contiene i metodi comuni a tutti gli oggetti dao.
  - DaoFactory : la classe astratta che permette la scelta del DBMS usato e fornisce i metodi astratti per la creazione degli oggetti dao.
  - MySqlDaoFactory : la classe che permette la creazione pratica degli oggetti dao implementando i metodi astratti ereditati da DaoFactory.

- interfaces

  contiene le interfacce specifiche per ogni dao che vanno ad estendere l'interfaccia base.BaseDao.

- implementation

  contiene l'implementazione effettiva del dao che consiste in delle classi astratte, Dao(Nome), (una per ogni oggetto dao) che implementeranno ogni una la corrispondente interfaccia interfaces.BaseDao(Nome). Ciascuna di queste classi astratte implementerà i metodi ereditati da interfaces.BaseDao(Nome) tranne getConnection(…) e cloaseConnection(…) e sarà estesa dalla classe implementation.Dao(Nome)MySql nella quale verranno implementati i due metodi di connessione appena citati. Ciò è stato fatto per fornire una visione più scarna e pulita del codice.

END